SMT-Based Verification of ECMAScript Programs in CPAchecker

Michael Maier

LMU Munich, Germany
JavaScript

- Most commonly used programming language\(^1\)
- Main language for Web applications
- Also used in server, desktop, and mobile applications
- Evaluated by interpreter
- Interpreter define different dialects

\(^1\)https://insights.stackoverflow.com/survey/2019/
ECMAScript

- Specified in standard ECMA-262\(^2\)
- Different standard versions
- Most\(^3\) JS dialects conform to ECMAScript 5.1\(^4\)

---

\(^2\)https://www.ecma-international.org/publications/standards/Ecma-262.htm

\(^3\)http://kangax.github.io/compat-table/es5/

\(^4\)https://www.ecma-international.org/ecma-262/5.1/
Goal

Extend **CPAchecker** to a restricted subset of ECMAScript 5.1 for SMT based verification approaches

- Parser frontend that creates CFA

- Strongest post operator (SMT formula encoding)
Restricted Subset Of ECMAScript 5.1

- No rarely used statements
- No recursive function calls
- No exceptions in general
- No standard built-in ECMAScript objects
Parser Frontend
Program Representation (CFA)
CFA Operations

- Assumption \[ p \]
- Variable declaration \( \text{var} \ x \) or \( \text{var} \ x = e \)
- Function declaration \( \text{function} \ \text{func} \ (\text{args}* \} \{ \ldots \} \)
- Assignment \( \text{lhs} = e \),
- Delete operation \( \text{delete} \ o.\text{propName} \) or \( \text{delete} \ o[e] \)
- Function call \( \text{func}(e*) \)
- Constructor call \( \text{new} \ \text{func}(e*) \)
SMT Formula Encoding
Strongest Post Operator

- CPAchecker does reachability analysis based on the CFA
- A location is reachable if a path \( \sigma \) to it exists, where \( \text{SP}_\sigma(\top) \) is satisfiable
- \( \text{SP}_\sigma(\psi) = \text{SP}_{op_m}(\ldots (\text{SP}_{op_i}(\psi)) \ldots) \)
- Goal of a verification task: Show that an error location is not reachable from the initial program location
- Definition of strongest post operator \( \text{SP}_{op}(\psi) \) required
Strongest post operator of an assumption $[p]$ as operator parameter:

$$\text{SP}_{[p]}(\psi) = \psi \land \text{ToBoolean}(p)$$

- $\text{ToBoolean}(p)$ represents large formula
- Other conversions like $\text{ToNumber}$, $\text{ToString}$, etc. have to be defined, too
Dynamic Types

```javascript
var x;
if (predicate) {
    x = 42;
} else {
    x = true;
}
var y = x;
```

- Variables may store values of different types
- SMT variables do not
Value ID

- SMT variables are used as value IDs (integer)\(^5\)
- Associate value and type with value ID using UFs\(^6\)
- Basic idea:
  - \(x = 42\) is encoded as
    
    \[
    \text{typeof}(x) = \tau_{\text{number}} \land \text{numberValue}(x) = 42
    \]
  - \(x = \text{true}\) is encoded as
    
    \[
    \text{typeof}(x) = \tau_{\text{boolean}} \land \text{booleanValue}(x) = \top
    \]

\(^5\)No assignment of specific integer required
\(^6\)UF = uninterpreted function
Static Single-Assingment Form

- Index counter is added for each variable
- Index counter is incremented on assignment
- Fresh value ID is used on every assignment

```javascript
var x;
if (predicate) {
    x = 42;
} else {
    x = true;
}
var y = x;

var x0;
if (predicate) {
    x1 = 42;
} else {
    x1 = true;
}
var y0 = x1;
```
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}

- On each call of `counter`
  - New variable `x` is created
  - New function object `next` is created
  - Variable `x` of `counter` is captured by reference from function object `next`
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
    c1(); // 2
    c1(); // 3
    c2(); // 1
    c1(); // 3

▶ c1 and c2 are different instances of next
▶ c1 and c2 do not capture the same x
Scoped Variables

- Use scoped value ID \( \text{var}(s, x) \)
- \( s \) is scope ID (integer)
- \( x \) is (unscoped) value ID
Scope Management

- On each function call
  - Create fresh scope ID
  - Put it on scope stack of called function object (array of scope IDs of outer function calls)
  - Associate it with this scope stack using UF `scopeStack`

- Track current scope using a variable

- On creation of function object associate it with current scope

- Captured variable `x` is encoded as

\[
\text{var}(\text{select}(\text{scopeStack}(\text{currentScope}), n), x)
\]

where `n` is the nesting level of the declaration of `x`
String values are mapped to a unique string ID by enumerating all constants of the program.

```javascript
var x = "foo";  // "foo" -> 1
var o = { bar: 42 };  // "prop" -> 2
o.foo: 42;  // "foo" -> 1
var t = typeof o;
// result of typeof may be:
// "undefined" -> 3, "object" -> 4, "boolean" -> 5,
// "number" -> 6, "string" -> 7, "object" -> 8,
// "function" -> 9
```
### Objects

```javascript
var o = {};
o.foo = 42;
```

- **Object value is represented by unique object ID (integer)**
  
  \[
  \text{typeof}(o) \land \text{objectValue}(o) = 1
  \]

- **Array** `objectFields` _i_ maps each object ID to its properties

  \[
  \text{store}(objectFields_i, 1, emptyObjectFields)
  \]

- **Property changes tracked by** `objectFields`

  \[
  fields_{old} = \text{select}(objectFields_i, 1) \\
  objectFields_{i+1} = \text{store}(objectFields_i, 1, fields_{new})
  \]
Properties

- Properties are managed as SMT array
- Property name (string ID) is mapped to value ID
- Names of unset properties are mapped to special value ID `objectFieldNotSet`
- For each set property use fresh value ID $p$ and mark it as set property
  \[ p \neq \text{objectFieldNotSet} \]
Example Of Property Changes

- Map all string IDs of the program to `objectFieldNotSet`.

```javascript
var o = {};
o.foo = 42;
o["foo"] = true;
o.bar = true;
delete o.foo;
```

<table>
<thead>
<tr>
<th>Program</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var o = {}</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>o.foo = 42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o[&quot;foo&quot;] = true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.bar = true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete o.foo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`objectFieldNotSet` is displayed as - in table.
Example Of Property Changes

- Use fresh value ID $p_0$ for `o.foo`

\[ p_0 \neq objectFieldNotSet \land typeof(p_0) = \tau_{number} \land numberValue(p_0) = 42 \]

<table>
<thead>
<tr>
<th>Program</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var o = {};</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>o.foo = 42;</td>
<td>$p_0$</td>
<td>-</td>
</tr>
<tr>
<td>o[&quot;foo&quot;] = true;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.bar = true;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete o.foo;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Of Property Changes

Use fresh value ID $p_1$ for next property assignment

```javascript
o["foo"] = true
```

$p_1 \neq \text{objectFieldNotSet}$

$\land \text{typeof}(p_1) = \tau_{boolean} \land \text{booleanValue}(p_1) = \top$

<table>
<thead>
<tr>
<th>Program</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var o = {};</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>o.foo = 42;</td>
<td>$p_0$</td>
<td>-</td>
</tr>
<tr>
<td>o[&quot;foo&quot;] = true;</td>
<td>$p_1$</td>
<td>-</td>
</tr>
<tr>
<td>o.bar = true;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete o.foo;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Of Property Changes

- Use fresh value ID $p_2$ for next property assignment

\[
o.bar = \text{true}
\]

\[
p_2 \neq \text{objectFieldNotSet} \\
\land \ \text{typeof}(p_2) = \tau_{\text{boolean}} \land \text{booleanValue}(p_2) = \top
\]

<table>
<thead>
<tr>
<th>Program</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var o = {};</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>o.foo = 42;</td>
<td>$p_0$</td>
<td>-</td>
</tr>
<tr>
<td>o[&quot;foo&quot;] = true;</td>
<td>$p_1$</td>
<td>-</td>
</tr>
<tr>
<td>o.bar = true;</td>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>delete o.foo;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Of Property Changes

Use `objectFieldNotSet` as value ID to `delete o.foo;`

<table>
<thead>
<tr>
<th>Program</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var o = {};</code></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>o.foo = 42;</code></td>
<td>$p_0$</td>
<td>-</td>
</tr>
<tr>
<td><code>o[&quot;foo&quot;] = true;</code></td>
<td>$p_1$</td>
<td>-</td>
</tr>
<tr>
<td><code>o.bar = true;</code></td>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td><code>delete o.foo;</code></td>
<td>-</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>
Prototype Chain Example

<table>
<thead>
<tr>
<th>Object</th>
<th>&quot;foo&quot;</th>
<th>&quot;bar&quot;</th>
<th>&quot;foobar&quot;</th>
<th>[[Prototype]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$p_0$</td>
<td>-</td>
<td>-</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>$p_1$</td>
<td>-</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>-</td>
</tr>
</tbody>
</table>

- If property is not found on object recursively look it up on prototype object (if existent)
- Prototype chain $a \rightarrow b \rightarrow c$

$^8$Special string ID for internal prototype property
Prototype Chain

- Prototype chain might be arbitrary long but it is always finite

- We assume that no prototype chain is longer as a maximum $maxPrototypeChainLength$

- Thereby, we can unroll the look-up in the prototype chain

- Drawback: look-up of a property might falsely return undefined if $maxPrototypeChainLength$ is too small
Evaluation
Approach

Evaluation of the functional correctness of the implementation of the formula encoding

- Using bounded model checking (BMC) with k-induction
- Based on the test programs of the official ECMAScript Conformance Test Suite Test262\(^9\)
- Automatic and manual filtering of test programs that contain unsupported or unimplemented features
- Generate negated tests (negate assertion condition) and check that they fail

\(^9\)https://github.com/tc39/test262
## Runs

Table: Results of different evaluation runs

<table>
<thead>
<tr>
<th>Run</th>
<th>Files</th>
<th>Correct</th>
<th>Incorrect</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>780</td>
<td>641</td>
<td>42</td>
<td>97</td>
</tr>
<tr>
<td>2</td>
<td>664</td>
<td>662</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8625</td>
<td>8593</td>
<td>13</td>
<td>19</td>
</tr>
</tbody>
</table>

1. Positive tests after automatic filtering
2. Positive tests after manual filtering and reconfiguring failed tests of 1st run
3. Negative tests of correct tests of 2nd run
Verification Results

- 660 test files correct (21 float encoded as rational)
- 2 test files unknown (timeout)
- 2 bugs\(^\text{10}\) found in Test262

\(^\text{10}\)https://github.com/tc39/test262/issues/2049
Solved Challenges

- Dynamic types
- Implicit type conversion
- Extensible objects
- Dynamic property access
- Prototype inheritance
- Function objects (higher order functions)
- Closures (scope chain)
Thank you for your attention!
Restricted Subset Of ECMAScript 5.1 (1/2)

- no recursive function calls
- no for-in statements
- no with statement
- no debugger statements
- no exceptions in general
- no standard built-in ECMAScript objects
- global variables are not set on the global object

- no arguments object

- it is assumed that all properties are named data properties that are writable and configurable

- implicit function calls from internal methods are not considered

- no regular expression literals

- no `<`, `>`, `<=`, and `>=` to compare strings
CFA Expressions

- **binary operators** & , / , == , === , > , >= , in , instanceof , << , < , <= , - , !== , != , | , + , % , >> , >>> , * , ~

- **unary operators** + , - , ∼ , ! , typeof , void

- **the property-access operators** o.f and o[p]

- **special operator** declaredBy (not part of regular ECMAScript)
declaredBy Operator

- used to resolve dynamic function calls

- `id declaredBy functionDeclaration` checks if the function object stored in `id` has been declared by the function declaration of `functionDeclaration`
Example of dynamic function call

```javascript
function f() { ... }
function g() { ... }
// ...
r = u();
```

- dynamic function call `u()`
- `f` or `g` might have been assigned to `u`
Resolution of dynamic function call

check by which function declaration the function object \( u \) has been declared and call that function

```javascript
function f() { ... }
function g() { ... }
// ...
// r = u();
if (u declaredBy f) {
  r = f();
} else if (u declaredBy g) {
  r = g();
} else {
  r = undefined; // this case would throw an exception,
  // but exceptions are not covered yet
```
Type Tags

Each type is encoded as a distinct integer called type tag (similar to the result of the `typeof` operator):

- $\tau_{\text{undefined}}$
- $\tau_{\text{boolean}}$
- $\tau_{\text{number}}$
- $\tau_{\text{string}}$
- $\tau_{\text{object}}$
- $\tau_{\text{function}}$
### Values

<table>
<thead>
<tr>
<th>ECMAScript Value</th>
<th>SMT-Formula Encoding Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>$\tau_{\text{undefined}}$</td>
<td>single value</td>
</tr>
<tr>
<td>Boolean</td>
<td>$\tau_{\text{boolean}}$</td>
<td>boolean</td>
</tr>
<tr>
<td>Number</td>
<td>$\tau_{\text{number}}$</td>
<td>$\text{FP}_{e=11,m=52}$ *</td>
</tr>
<tr>
<td>String</td>
<td>$\tau_{\text{string}}$</td>
<td>$\text{FP}_{e=12,m=52}$ *</td>
</tr>
<tr>
<td>Object</td>
<td>$\tau_{\text{object}}$</td>
<td>integer</td>
</tr>
<tr>
<td>Null</td>
<td>$\tau_{\text{object}}$</td>
<td>integer</td>
</tr>
<tr>
<td>Function</td>
<td>$\tau_{\text{function}}$</td>
<td>integer</td>
</tr>
</tbody>
</table>

* Floating point formula with exponent size $e$ and mantissa size $m$
string values are mapped to a unique string-ID (floating point number)

string values that are strict equal\(^{11}\) have the same ID

string-IDs are encoded as $\text{FP}_{e=12,m=52}$

ECMAScript number values are encoded as $\text{FP}_{e=11,m=52}$

values in range $\text{FP}_{e=11,m=52}$ are used for string representations of their respective ECMAScript number value

values outside this range are used for all other strings

\(^{11}\)same sequence of characters
Function Object

- each function object value is encoded like a regular object, but its type is $\tau_{\text{function}}$

- its object-ID is associated with its
  - function declaration using an uninterpreted function declarationOf
  - scope using an uninterpreted function scopeOf
Value ID

- Associated type and value UF have to be compatible

\[
\text{typeof}(x) = \tau_{\text{boolean}} \land \text{numberValue}(x) = 42
\]

- Value ID may not be associate with different types

\[
\text{typeof}(x) = \tau_{\text{boolean}} \land \text{typeof}(x) = \tau_{\text{number}}
\]
Different values are associated with different value IDs

\[
\text{typeof}(x) = \tau_{\text{number}} \land \text{numberValue}(x) = 42 \\
\land \text{typeof}(y) = \tau_{\text{boolean}} \land \text{booleanValue}(y) = \top
\]

or the same value ID using mutually exclusive conditions

\[
(p \land \text{typeof}(x) = \tau_{\text{number}} \land \text{numberValue}(x) = 42) \\
\lor (\neg p \land \text{typeof}(x) = \tau_{\text{boolean}} \land \text{booleanValue}(x) = \top)
\]
Scope Management

- Create scope ID for `currentScope` on function call
- Put it on its scope Stack

```javascript
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}

var c1 = counter();
var c2 = counter();
c1();  // 1
   c1();  // 2
   c1();  // 3
   c2();  // 1
   c2();  // 1
```

`currentScope = 1`

`scopeStack(1) = (1)`
Scope Management

- Use `currentScope` for local variables of current call

```javascript
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
    c1(); // 2
    c1(); // 3
```

`currentScope = 1`
`scopeStack(1) = (1)`
`numberValue(var(1, x0)) = 0`
Create function object and associate it with `currentScope`

```javascript
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}

var c1 = counter();
var c2 = counter();
c1(); // 1
    c1();  // 2
        c1();  // 3
Scope Management

- Create scope ID for `currentScope` on function call
- Put it on its scope Stack

```javascript
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
```

```
currentScope = 2
scopeStack(1) = (1)
scopeStack(2) = (2)
numberValue(var(1, x₀)) = 0
scopeOf(c1) = 1
```
Scope Management

- Use `currentScope`, update other scoped variables

```javascript
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}

var c1 = counter();
var c2 = counter();
c1(); // 1

var(1, x_0) = 0

var(2, x_1) = 1

c1(); // 2

var(1, x_1) = var(1, x_0)

scopeOf(c1) = 1

var(2, x_1) = (2)

scopeStack(2) = (2)

numberValue(var(2, x_1)) = 0

currentScope = 2

scopeStack(1) = (1)

numberValue(var(1, x_0)) = 0

currentScope = 2
```

Michael Maier
LMU Munich, Germany
Scope Management

- Simplify formulas

```javascript
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}

var c1 = counter();
var c2 = counter();
c1(); // 1
    c1(); // 2
    c1(); // 3
  c2(); // 1
  c1(); // 1
  c1(); // 2

currentScope = 2
scopeStack(1) = (1)
scopeStack(2) = (2)
numberValue(var(1, x1)) = 0
numberValue(var(2, x1)) = 0
scopeOf(c1) = 1
```
Scope Management

- Create function object and associate it with `currentScope`

```javascript
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}

var c1 = counter();
var c2 = counter();
c1(); // 1
var c1 = counter();
c1(); // 2
c2(); // 1
c1(); // 3

currentScope = 2
scopeStack(1) = (1)
scopeStack(2) = (2)
numberValue(var(1, x1)) = 0
numberValue(var(2, x1)) = 0
scopeOf(c1) = 1
scopeOf(c2) = 2
```
Scope Management

- Create scope ID for currentScope on function call
- Put it on the scope Stack of the called function object

```javascript
function counter() {
    var x = 0;
    return function next() {
        x = x + 1;
        return x;
    }
}
var c1 = counter();
var c2 = counter();
c1(); // 1
    // 1
    // 2
    // 1
    // 3
```

```
currentScope = 3
scopeStack(1) = (1)
scopeStack(2) = (2)
scopeStack(3) = (1, 3)
numberValue(var(1, x)) = 0
numberValue(var(2, x)) = 0
scopeOf(c1) = 1
scopeOf(c2) = 2
```
Scope Management

- Use \( \text{var}(\text{select}(\text{scopeStack}(<currentScope>), n), x_i) \)

- Results in \( \text{var}(1, x_i) \)

```javascript
function counter() {
  var x = 0;
  return function next() {
    x = x + 1;
    return x;
  }
}

var c1 = counter();
var c2 = counter();
c1(); // 1
  // 1
  // 2
  // 1
  // 3
```

\( currentScope = 3 \)
\( \text{scopeStack}(1) = (1) \)
\( \text{scopeStack}(2) = (2) \)
\( \text{scopeStack}(3) = (1, 3) \)
\( \text{numberValue}(\text{var}(1, x_1)) = 0 \)
\( \text{numberValue}(\text{var}(1, x_2)) = 1 \)
\( \text{scopeOf}(c1) = 1 \)
\( \text{scopeOf}(c2) = 2 \)
Variable Declaration

- `var x` is handled like an assignment operation
  ```javascript
  x = undefined
  ```

- `var x = e` is handled like an assignment operation
  ```javascript
  x = e
  ```
function declaration  

```
function func(args*) { ... }
```

function object of `func` is created similar to

```
{
  prototype: {},
  length: len
}
```

where `len` represents the count of function parameters
Function Declaration (2/2)

function declaration

```javascript
function func(args*) { ... }
```

- object-ID \( o \) of the created object is used in constraints

\[
\begin{align*}
typeof(fv) &= \tau_{\text{function}} \\
functionValue(fv) &= o \\
objectValue(fv) &= o \\
scopeOf(o) &= currentScope \\
declarationOf(o) &= d
\end{align*}
\]

where

- \( fv \) is the scoped variable of the function declaration identifier `func`
- \( d \) is the declaration-ID of the declared function
Assignment

different assignment targets:

- assignment to identifier \( x = e \)
- assignment to object property
  - dot notation \( \text{obj.propName} = e \)
  - bracket notation \( \text{obj[propExpr]} = e \)
Assignment To Identifier

assignment to identifier \( x = e \)

- associate type of variable with type of expression
- for each type case assign the respective value
- update other scoped variables (same declaration, but different scope-ID)
Assignment To Object Property

\[
\text{obj}.\text{propName} = e \quad \text{or} \quad \text{obj}[\text{propExpr}] = e
\]

- create a fresh variable-ID \( p \) and mark it as set property (variable)
  \[
  p \neq \text{objectFieldNotSet}
  \]
- assign value to \( p \)
- associate property of object with \( p \)
- in case of bracket operator, ensure update of \text{length} property (another property assignment)
Delete Operator

```
delete operation  delete o.propName or delete o[e]
```

- equivalent to assigning `objectFieldNotSet` to the property of the object that is deleted
Function Call

Function call \( func(e*) \)

- execution context switches from the *caller* (function or global code) to the *called* function

- the following has to be done:
  - create a new scope for the called function
  - update current scope stack
  - bind (optional) this argument
  - assign arguments of call to parameter variables of called function
Constructor Call

constructor call  `new func(e*)`

- handled like a function call  `func(e*)`

- but a new object is created and assigned to the this variable:

```
{
    [[Prototype]]: func.prototype
}
```

where  `[[Prototype]]`  represents the prototype property
## Prototype Chain Example

<table>
<thead>
<tr>
<th>Object</th>
<th>Property Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>‘foo’</td>
</tr>
<tr>
<td>A</td>
<td>-</td>
</tr>
<tr>
<td>A.prototype</td>
<td>foo₂</td>
</tr>
<tr>
<td>B</td>
<td>foo₂</td>
</tr>
<tr>
<td>bProto / B.prototype</td>
<td>foo₂</td>
</tr>
<tr>
<td>b</td>
<td>foo₂</td>
</tr>
</tbody>
</table>

```javascript
function A() {}
A.prototype.foo = 1;
function B() {}
var bProto = new A();
bProto.bar = 2;
B.prototype = bProto;
var b = new B();
```
Implementation
Options

- **Maximum Field Count:** Required to initially map all properties to `objectFieldNotSet`

- **Maximum Prototype Chain Length:** Required to unroll the prototype chain

- **Usage Of NaN and infinity:**
  - Rational formulas do not support the values `NaN` and $\pm\infty$ (only as a variable)
  - If floating point formulas are encoded as rational formulas, checking for `NaN` or $\pm\infty$ can lead to satisfiable and non-tautological formulas
  - Option alters the formula encoding by assuming that those checks always result in $\bot$